

Circuit translates A law to μ law

ROLANDO HERRERO, INSTITUTO TECNOLÓGICO DE BUENOS AIRES, ARGENTINA

Two common methods exist to compand voice for transmission through a PCM channel. In Europe, A law involves converting a 12-bit input signal to an 8-bit encoded output. In the US, μ law involves encoding 13 bits to 8 bits. You can use a translator to convert from A law to μ law (Figure 1). The converter is asynchronous and requires only an 8-bit A law input to provide an 8-bit μ law output.

In A law, the input level divides into eight regions in which a uniform 4-bit conversion takes place. Regardless of the region, the output encodes 16 possible values. Each region corresponds to a segment in Figure 2, and the lower values have a better resolution (this figure shows only segments 0 through 5). To encode the input takes 8 bits; 4 bits indicate the uniform converted value in the segment, and the other 4 bits divide to represent the segment value itself (S0 to S7, coded with 3 bits) and whether the signal is positive or negative (1 bit).

Alternatively, with μ law, also included in Figure 2, all but the first segments have a wider dynamic range and thus more spaced quantization levels (for 4 bits) compared with A law. Instead of 12 bits, 13 bits imply a wider dynamic range but a worse resolution for low input levels.

Figure 2 also illustrates the loss of resolution when converting the output A of A law to output A' of μ law. Depending on the law, either 8 bits (A law) or 4 bits (μ law and higher quantization levels) represent the value, therefore, the transitions occur faster around output A than around output A'. For the A to A' translation, the slope of A law is twice the slope of μ law.

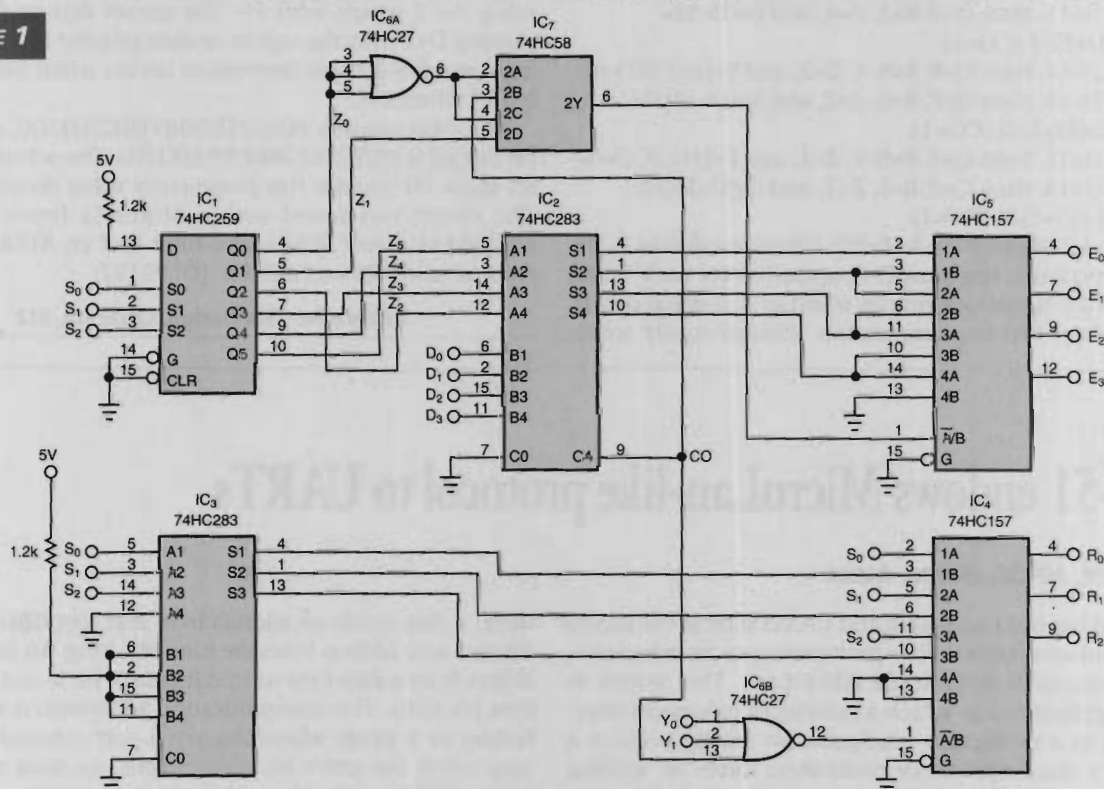
Although information loss occurs during the conversion of value A, the same is not true for B. For B, the A law and μ law slope are the same, and the quantization level is the same. Thus, the difference between B and B' involves only a translation and a change of segment (B in S4, B' in S3). A simple comparison shows that the A value suffers a translation and a loss of information but remains in the same segment after conversion.

The design of the encoder must take into account the A law signal's segment and offset value, as does the following algorithm for which the A law input signal is PSD, and the μ law output signal is QRE, for which P,Q=polarity (1 bit), S,R=segment (3 bits) and D,E=value (4 bits):

If S=0, then Q=P, R=S, and E=D.

If S=1, then Q=P, R=S, and E=D/2.

FIGURE 1



This A law-to- μ law translator inputs values of S and D and outputs E and R according to a specific algorithm.

If $S=2$ and $D<8$, then $Q=P$, $R=S-1$, and $E=D+8$.

If $S=2$ and $D>7$, then $Q=P$, $R=S$, and $E=(D-8)/2$.

If $S=3$ and $D<12$, then $Q=P$, $R=S-1$, and $E=D+4$.

If $S=3$ and $D>11$, then $Q=P$, $R=S$, and $E=(D-12)/2$.

If $S=4$ and $D<14$, then $Q=P$, $R=S-1$, and $E=D+2$.

If $S=4$ and $D>13$, then $Q=P$, $R=S$, and $E=(D-14)/2$.

If $S=5$ and $D<15$, then $Q=P$, $R=S-1$, and $E=D+1$.

If $S=5$ and $D>14$, then $Q=P$, $R=S$, and $E=(D-15)/2$.

If $S=6$, then $Q=P$, $R=S-1$, and $E=D$.

If $S=7$, then $Q=P$, $R=S-1$, and $E=D$.

According to this algorithm, the conversion requires both addition and subtraction, depending on S and D . You can express each subtraction as an addition to implement both in the same circuit. Thus, you can express the algorithm as follows, where $CO=$ Carry out:

If $S=2$ and $D<8$, then $Q=P$, $R=S-1$, $Z=8$, and $E=D+Z$ ($CO=0$).

If $S=2$ and $D>7$, then $Q=P$, $R=S$, $Z=8$, and $E=(D-8)/2=(D-16+Z)/2=(D+Z)/2$ ($CO=1$).

If $S=3$ and $D<12$, then $Q=P$, $R=S-1$, $Z=4$, and $E=D+Z$ ($CO=0$).

If $S=3$ and $D>11$, then $Q=P$, $R=S$, $Z=4$, and $E=(D-12)/2=(D-16+Z)/2=(D+Z)/2$ ($CO=1$).

If $S=4$ and $D<14$, then $Q=P$, $R=S-1$, $Z=2$, and $E=D+Z$ ($CO=0$).

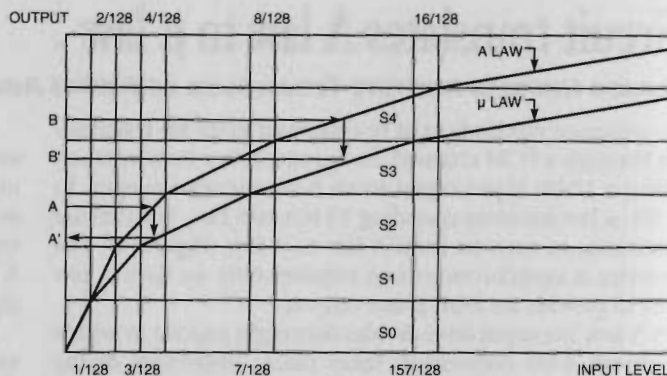
If $S=4$ and $D>13$, then $Q=P$, $R=S$, $Z=2$, and $E=(D-14)/2=(D-16+Z)/2=(D+Z)/2$ ($CO=1$).

If $S=5$ and $D<15$, then $Q=P$, $R=S-1$, $Z=1$, and $E=D+Z$ ($CO=0$).

If $S=5$ and $D>14$, then $Q=P$, $R=S$, $Z=1$, and $E=(D-15)/2=(D-16+Z)/2=(D+Z)/2$ ($CO=1$).

The value of Z depends on S : $Z=2^{5-S}$. Once you define Z , the algorithm performs the same $D+Z$ operation for each S . The carry-out (CO) signal determines whether R is equal to S or $S-1$. Therefore, this implementation simultaneously solves

FIGURE 2



Converting output A of A law to A' of μ law incurs a loss of information. However, no information loss occurs when converting from B to B', because the slopes of the two curves are the same at that point.

two problems. Furthermore, the same technique applies for $S=6$ and $S=7$, when $Z=0$.

In Figure 1, a 3×8 decoder, IC_1 , converts S to Z , which IC_2 adds to D . If the CO is a 1, E is $(D+Z)/2$; otherwise, R is $S-1$. To choose between both options, the circuit uses the CO signal to control data selectors IC_4 and IC_5 . These devices select between two possible outputs: S or $S-1$ and $D+Z$ or $(D+Z)/2$, respectively. A second adder, IC_3 , implements $S-1$ by summing the S inputs with 15. The circuit derives $(D+Z)/2$ by shifting $D+Z$ into the inputs of data selector IC_5 . Additional logic ensures that no conversion occurs when $S=0$ and that $E=D/2$ when $S=1$.

The 8-bit input is $P0/S2/S1/S0/D3/D2/D1/D0$, and the 8-bit output is $P0/R2/R1/R0/E3/E2/E1/E0$. The schematic doesn't show $P0$ because this parameter's value doesn't change. The circuit was tested with a Motorola (www.mot.com) MC145554 μ law PCM codec-filter and an 8TR641 (AT&T, www.att.com) E1 multiplexer. (DI #2192)

EDN

To Vote For This Design, Circle No. 412

MCS-51 endows MicroLan-like protocol to UARTs

SK SHENOY, NPOL, KOCHI, INDIA

μ Cs such as the 8051 and 8096 and UARTs such as the 82510 provide hardware support for a multiprocessor asynchronous serial-communication protocol (MicroLan). This feature is useful in applications in which a number of processors interconnected in a multipoint configuration jointly perform a task, with a master processor controlling slaves by sending data or commands in a selective manner (Figure 1). The protocol operates as follows:

When the master wishes to transmit a block of data to a

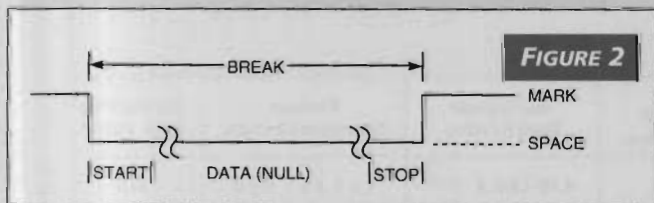
slave, it first sends an address byte that identifies the slave. All data and address bytes are nine bits long. An address byte differs from a data byte in that its ninth bit is one (for a data byte it's zero). The communication subsystem normally initializes in a mode where the serial-port interrupt activates only when the ninth bit is one. Thus, no slave receives an interrupt from a data byte. An address byte, however, interrupts all slaves, which then examine the received byte. Next, the addressed slave switches to a mode in which data bytes

also receive interrupts, while other slaves go about their business uninterrupted by the data transfer. The address bytes thus control the data flow into a particular node. Indication of the end of a data block can come from either sending a data-length field at the beginning of the block or from the receipt of another slave or reserved address.

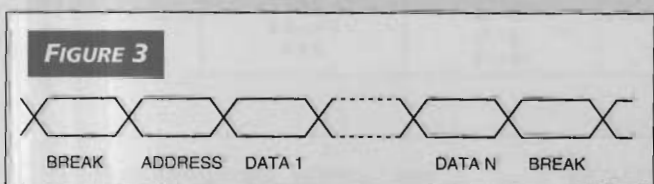
The crucial requirement for realizing the protocol is a means of distinguishing address from data bytes. You can effect this identification in many popular UARTs by using an obscure feature found in most UARTs: the capability to transmit and recognize (with an interrupt on) the break condition. This condition is nothing but a "space," or low, in the transmit line, of a duration equal to or greater than an entire asynchronous character-transmission time, including stop bits (Figure 2). In this scheme, the whole data block (including address) from a master is sandwiched between break characters to form a data "frame" (Figure 3), and the address byte is recognizable as the one that immediately follows a break character.

The Turbo C program in Listing 1 demonstrates the transfer of variable-size messages between two PCs (with 8250-compatible UARTs) using the method described here. Figure 4 shows the 8250 register formats. The procedure works with most other UARTs. You can download the file from EDN's Web site, www.ednmag.com. At the registered-user area, go into the "Software Center" to download the listing from DISIG #2193. A null-modem cable interconnects the PCs' COM ports. The destination PC accepts only the messages addressed to it. Note that, although the PCs here interconnect in a point-to-point manner, usually the stations interconnect using balanced RS-422 or tristate drivers in a multipoint configuration, as in Figure 1.

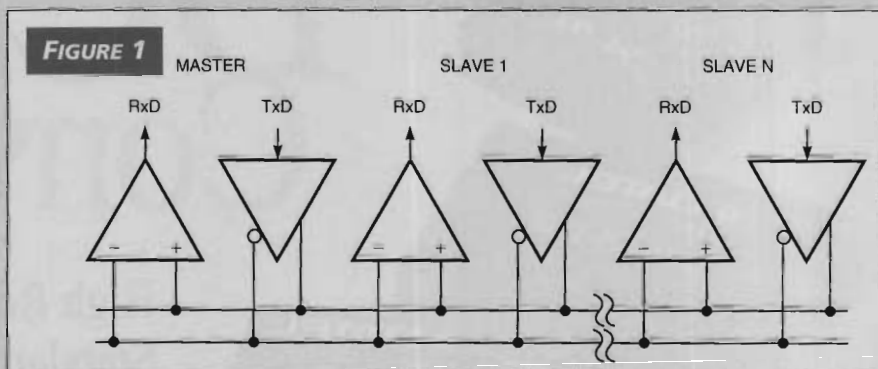
A global variable, `Receive_Count`, initialized to zero, han-



The ability to recognize the break condition is key to the master-slave transfer protocol.



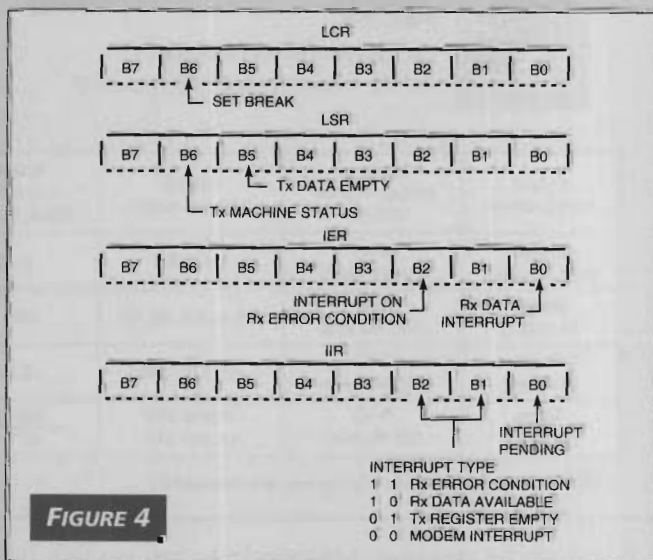
This n-byte data frame shows the data block sandwiched between break characters.



A master-slave arrangement uses RS-422 transceivers to effect a multipoint data-transfer configuration.

dles frame reception. Initially, the protocol enables only receive-error interrupts. Each time the routine detects a break, the UART raises a receive-error interrupt, and the ISR (interrupt service routine) then enables the receive-data interrupts. On subsequent receive interrupts, if `Receive_Count` is zero, the ISR checks if the first address byte matches the station's address. If not, the receiver goes back to the initial waiting state, with the receive-error interrupts enabled and the receive-data interrupts disabled, such that the routine ignores the subsequent data bytes. If an address match occurs, the ISR stores the subsequent incoming data bytes in the receive buffer, with `Receive_Count` as index. If `Receive_Count` is nonzero when the break interrupt occurs, it is an end-of-frame break. Then the routine calls the frame-processing function, `Receive_Count` resets to zero, and the receiver again reverts to the initial waiting state.

To transmit a break, the protocol sets bit 6 (set break) of the line-control register to one. The UART then takes its trans-



These 8250 register formats demonstrate the multipoint-transfer protocol.

mission line low until bit 6 receives a zero. To make the duration of the break equal to one character-transmission delay, the routine transmits a null (00 hex) character. Bit 6 of the line-control register (transmit machine status) indicates when this delay is over; the break bit then resets. To enable detection of the break, bit 2 of the interrupt enable register (interrupt on receive error condition) sets during UART initialization. Bit 0, set to one, enables receive data interrupts. In the ISR, bits 1 and 2 of the interrupt-identification register indicate the interrupt type.

In this scheme, no CPU overhead is wasted examining each character to detect addresses/packet boundaries. Also, a slave must process only three interrupts per data packet transmit-

ted on the bus, and blocks of data not addressed to the slave do not disturb it. Because the break is not a legitimate data character, it is data transparent; you can use it for binary-data exchange. The packet-boundary detection is immune to data errors. You can make it even more robust by including data-length and check-sum fields in the frame to enable error detection. You can also use parity error detection. Note that the method can support broadcast/multicast message transfer by designating some addresses for these purposes. You can also implement any-node-to-any-node communication by polling the master, as in the SDLC protocol. (DI #2193) **EDN**

To Vote For This Design, Circle No. 413

LISTING 1—TRANSFER OF VARIABLE-SIZE MESSAGES BETWEEN TWO PCs

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

/* COM PORT DEFINITIONS AND GLOBAL VARIABLES */
#define com_reg 0x3f8 /* Default is com1, 2f8 for com2 */
#define DATA_PORT com_reg + 0
#define LINE_CNTRL com_reg + 3
#define MODEM_CNTRL com_reg + 4
#define INT_ENABL com_reg + 1
#define INT_IDENT com_reg + 2
#define LINE_STS com_reg + 5
#define MODEM_STS com_reg + 6
#define BAUD_LOW com_reg + 0
#define BAUD_HIGH com_reg + 1
#define DLAB_SET com_reg + 0x00
#define BAUDMSB 0
#define BAUDLSB 0xc /* 9600 bps */
#define CNTRL_CMD 7 /* 8 BIT, 2 STOP BIT, NO PARITY */
#define WAIT_TX_RDY() while (((inportb(LINE_STS)&0x50)!=0x50)

/* Check for Tx buf empty & Tx shift reg empty */

unsigned char rdbuf[256], rdbuf[256]; /* Send & Recv buffers */
int Receive_Count = 0; /* Counter for data stored in rdbuf[] */
void interrupt (*OldComHandler) (void);
unsigned char Myaddr, Txaddr;

void processdata(void) /* TO DISPLAY RECEIVED DATA PACKET */
{
    int i;
    printf("\nRX Data > "); clrscr(); /* Received data cursor */
    for (i = 1; i < Receive_Count; i++) /* Leaving out Addr byte */
        putchar(rdbuf[i]); /* Display received data */
    printf("\n"); /* New line */
    clrscr(); /* Clear line */
}

void interrupt service_sio(void) /* ISR: TAKES CARE OF PACKET RECEPTION */
{
    unsigned char iir;

    iir = (inportb(INT_IDENT) >> 1) & 3; /* Get interrupt type */
    switch(iir)
    {
        case 0: /* Modem status int DSR, CTS, RI, RLSD */
            inportb(MODEM_STS); /* Ignore; reading IIR resets int */
            break; /* reading IIR resets int */
        case 1: /* Tx int */
            break; /* reading IIR resets int */
        case 2: /* Rx int */
            rdbuf[Receive_Count++] = inportb(DATA_PORT); /* Store packet data */
            if ((Receive_Count == 1) && (rdbuf[0] != Myaddr))
                /* If First(Address) byte but no address match */
            {
                outportb(INT_ENABL, 0x4); /* IER; enable Only Rx Machine error int */
                Receive_Count = 0;
            }
            break;
        case 3: /* Rx error (Break detect etc.) */
            inportb(DATA_PORT); /* Read Null char */
            if ((inportb(LINE_STS)&0x10) == 0x10)
                /* Break detected; Reading LSR Resets int */
            {
                if (Receive_Count) /* Complete Frame Over */
                {
                    processdata(); /* Process the frame */
                    outportb(INT_ENABL, 0x4); /* IER; enable only Rx Machine error int */
                }
                else outportb(INT_ENABL, 0x5); /* IER; enable RX Data int also */
                Receive_Count = 0; /* Reinitialize for next frame */
            }
            outportb(0x20, 0x20); /* EOI to 8259 PIC */
            return;
    }

    void init_serial_io(void) /* TO INITIALISE SERIAL PORT */
{
    outp(LINE_CNTRL, DLAB_SET); /* DLAB SET */
    outp(BAUD_LOW, BAUDLSB); outp(BAUD_HIGH, BAUDMSB); /* 9600 BAUD */
    outp(LINE_CNTRL, CNTRL_CMD); /* 8 BIT, 2 STOP BIT, NO PARITY */
    outp(MODEM_CNTRL, 8); /* DTR, RTS & OUT2 SET */
    OldComHandler = getvect(0xc); /* 0xb for com2 */
    disable();
    setvect(0xc, (service_sio)); /* 0xb for com2 */
    outportb(0x21, ((inportb(0x21)&0x10)); /* PIC mask word 0x8 for com2 */
    outportb(INT_ENABL, 0x4); /* IER; enable only Rx Machine error int */
    enable();
}

void SendBreak(void) /* TO TRANSMIT A BREAK OF ONE CHARACTER DURATION */
{
    outportb(LINE_CNTRL, inportb(LINE_CNTRL) | 0x40); /* LCR; set break */
    outportb(DATA_PORT, 0); /* Send NULL data */
    WAIT_TX_RDY(); /* Wait on TxShift Reg Empty; Null char is shifted out */
    outportb(LINE_CNTRL, inportb(LINE_CNTRL) & 0xbf); /* LCR; remove break */
}

/* TO TRANSMIT A DATA PACKET */
void SendBuffer(unsigned char packet[], int DatLen)
{
    int i;
    SendBreak(); /* Send START OF PACKET break */
    WAIT_TX_RDY(); /* Wait for Tx Ready */
    outportb(DATA_PORT, Txaddr); /* Send Tx address */
    for (i = 0; i < DatLen; i++) /* For each message byte */
    {
        WAIT_TX_RDY(); /* Wait for Tx Ready */
        outportb(DATA_PORT, packet[i]); /* Send next data char */
    }
    WAIT_TX_RDY(); /* Wait for Tx Ready */
    SendBreak(); /* Send END OF PACKET break */
    WAIT_TX_RDY(); /* Wait for Tx Ready */
}

unsigned char getaddr(char* mess) /* TO READ AN ADDRESS FROM THE CONSOLE */
{
    unsigned char c, databuf[100];
    int addr, count = 0;

    printf("\n\nr%s", mess); /* Prompt for input */
    clrscr();
    while(1) /* Forever Loop */
    {
        if ((c = getch()) == 27) exit(0); /* Exit if Escape key pressed */
        databuf[count++] = c; /* Get typed characters into the buf */
        if (c == '\r') /* If Enter Key pressed */
        {
            if (!scanf(databuf, "%d", &addr) || ((addr > 0xff) || (addr < 0)))
                /* Read and check the string in the buffer for validity */
            {
                putchar(7); /* Bell */
                printf("\n\nError: Type in a number between 0 and 255");
                printf("\n\nr%s", mess); /* Transmit Prompt */
                clrscr(); /* Clear to end of line */
                count = 0;
            }
            else break;
        }
    }
    return((unsigned char)addr);
}

void restoreint(void) /* FUNCTION WHICH DOES THE CLEAN-UP AT EXIT TIME */
{
    setvect(0xc, OldComHandler); /* Restore int vector; 0xb for com2 */
    outportb(0x21, ((inportb(0x21)&0x10)); /* PIC mask word 0x8 for com2 */
}
```

Voltage monitor prevents deep discharge of battery

ROGER KENYON, MAXIM INTEGRATED PRODUCTS, SUNNYVALE, CA

The circuit in **Figure 1** monitors battery voltages from 2.7 to 5.5V while drawing less than 25 μ A. When the voltage reaches a minimum threshold established by R_1 and R_2 , ($V_{TH}=2.63V$ for the values shown), the high-side switch (IC_2) turns off and disconnects the battery from the load.

IC_1 is a voltage monitor with an open-drain latched output. Normally high, the output latches low when the battery voltage drops below V_{TH} . Once triggered, the output remains low even when the now-unloaded battery voltage rebounds to a level above V_{TH} . This behavior prevents the oscillation that would otherwise occur as connect/disconnect action causes the battery voltage to fluctuate. To reset the latch, the CLEAR input must go high for a minimum of 1 μ sec.

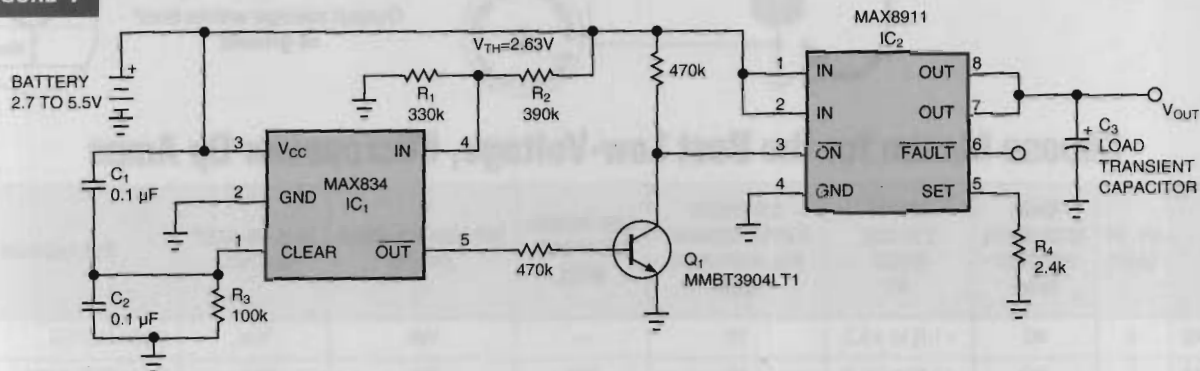
The $C_1/C_2/R_3$ network applies the latch-clearing pulse when you connect a new battery. Rechargeable-battery applications require other schemes for clearing the IC_1 output, such as an spst momentary pushbutton switch (**Figure 2a**) or simply a connection via the battery-charger connector (**Figure 2b**).

To set a different value of V_{TH} , choose a convenient value for R_1 , and then calculate R_2 : $R_2=R_1 \times V_{TH}/(1.204-1)$. IC_2 limits its switch current at a level that the value of R_4 determines: $I_{LIMIT}=1240/R_4$, where R_4 is in ohms and I_{LIMIT} is in amperes, with a maximum of 1A. For the R_4 value in **Figure 1**, this limit is 500 mA. (DI #2191)

EDN

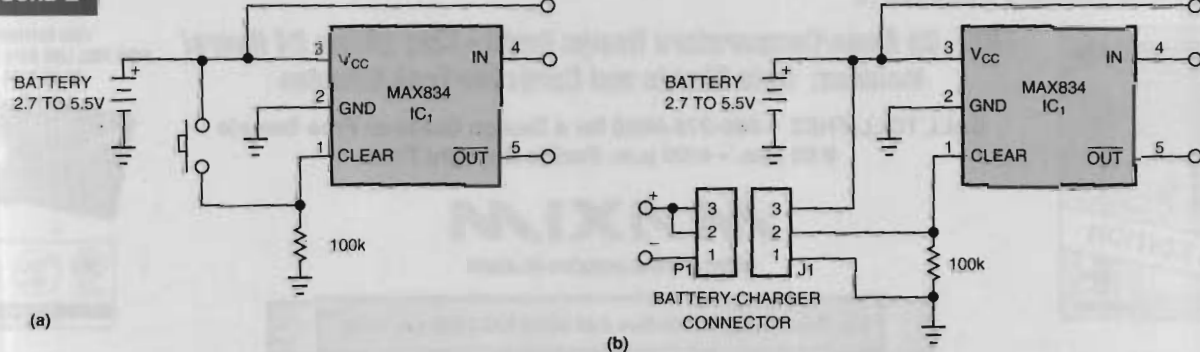
To Vote For This Design, Circle No. 414

FIGURE 1



When the voltage reaches a minimum threshold established by R_1 and R_2 , the high-side switch, IC_2 , turns off and disconnects the battery from the load.

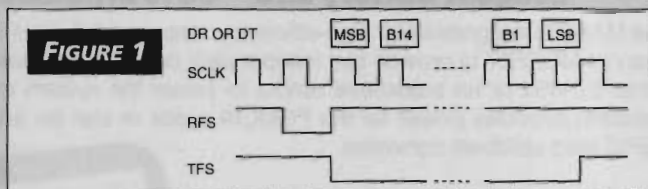
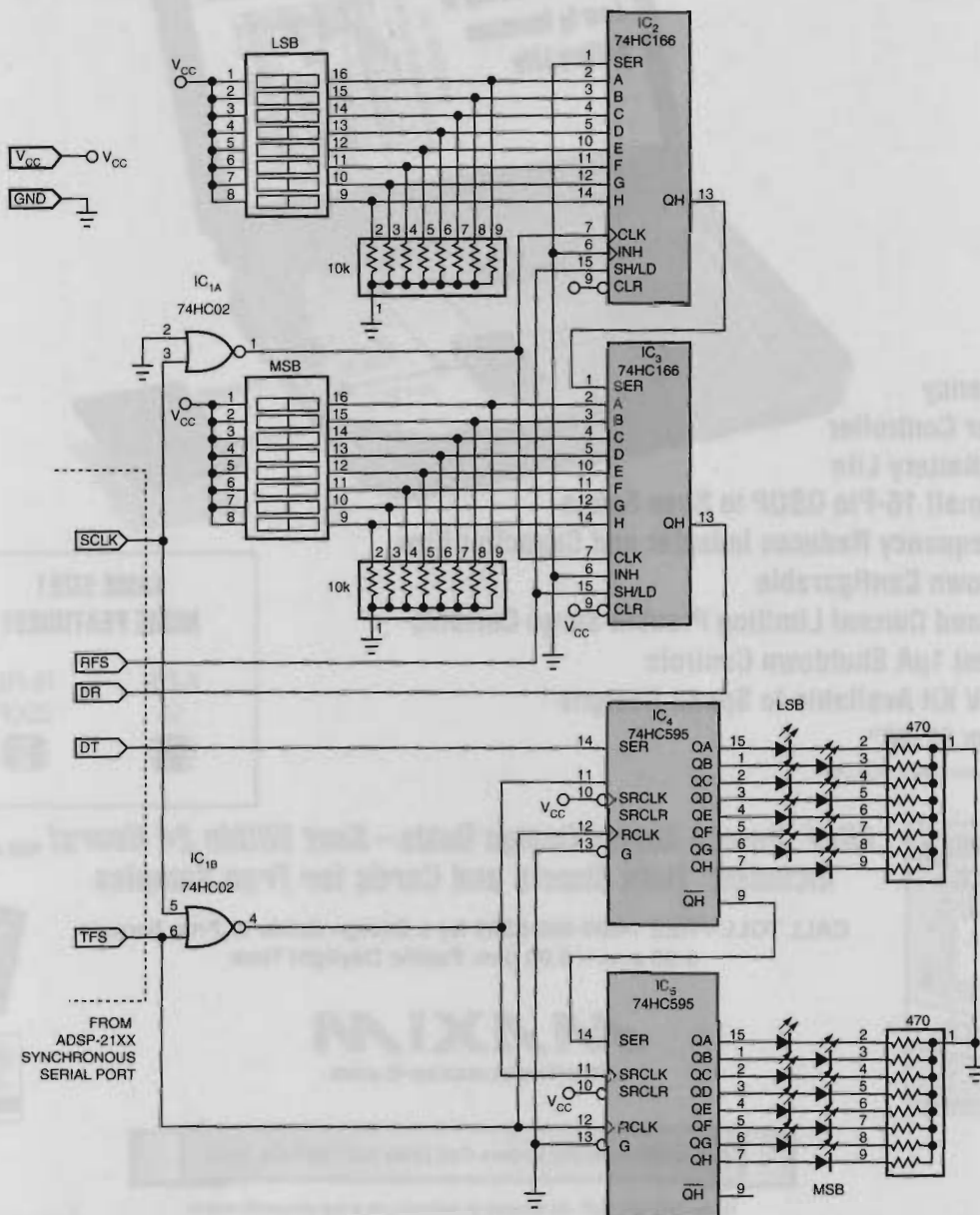
FIGURE 2



Other schemes for clearing IC_1 's output include an spst momentary pushbutton switch (a) and a connection through the battery-charger connector (b).

STAN SASAKI, TWENTY-FIRST DESIGNS, LAKE OSWEGO, OR

A bank of switches and LEDs simulates inputs and displays intermediate results as an aid to debugging embedded-DSP designs.

**FIGURE 2**

136 • EDN MAY 7, 1998

LEDs to the data bus using octal latches and decoded address strobes. However, this connection can be a wiring nightmare—or even impossible if the design does not generate the required control strobes. Most DSP chips have a synchronous serial port that provides high-speed (>10 Mbps) bidirectional communication over five wires. The circuit in **Figure 1** provides 16-bit switch and LED I/O using the on-chip serial port of Analog Devices' ADSP21xx family. Some ADSP devices, such as the ADSP2105, have only a single serial port that the target application can use. However, during debug, you can use the pin-compatible ADSP2115, which has two serial ports. You can bring the five signals from the second serial port to a header that attaches to the circuit in **Figure 1**.

The DSP generates a continuous serial clock (SCLK) in **Figure 2**. Data shifts, MSB-first, in each direction and is valid on the falling edge of SCLK. For transmission to the LEDs, the DSP asserts the transmit-frame-sync (TFS) line while the 16 bits clock out on DT (data transmit). Cascaded 8-bit serial-to-parallel shift registers, IC₄ and IC₅, capture the data, and the LEDs receive an update on the rising edge of TFS. For switch reception, the DSP generates a receive-frame-sync (RFS) pulse one bit-time before reading the data. RFS latches the switch

states into cascaded parallel-to-serial shift registers, IC₂ and IC₃, and the data shifts into the serial port over DR (data receive) on the next 16 SCLK cycles. You can configure the serial port to generate the RFS pulse at a rate divided down from the SCLK frequency.

A single instruction reads the switches or writes to the LEDs. For example, the instruction AR=RXn transfers the last switch value from the receiver of serial port n into the 16-bit accumulator, AR. Note that n can be either 0 or 1, depending on which on-chip serial port you use. Similarly, the instruction TXn=AR transfers the 16-bit value in AR to the transmitter of serial port n, and on to the LEDs. You must configure the serial port to operate with the frame-sync types and signal polarities the circuit expects. In this case, the 16-bit value you must write to the serial port n control register is 6FCF. Because 16 LEDs can draw more than 100 mA, you may need to provide external power if the circuit under test cannot supply the necessary current. Because HC parts operate at 3 and 5V, this circuit works for both 3 and 5V ADSP devices. However, for 3V operation you may need to lower the SCLK speed to meet HC performance. (DI #2196)

EDN

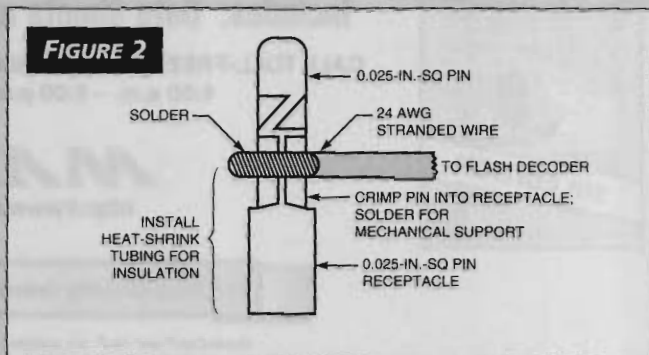
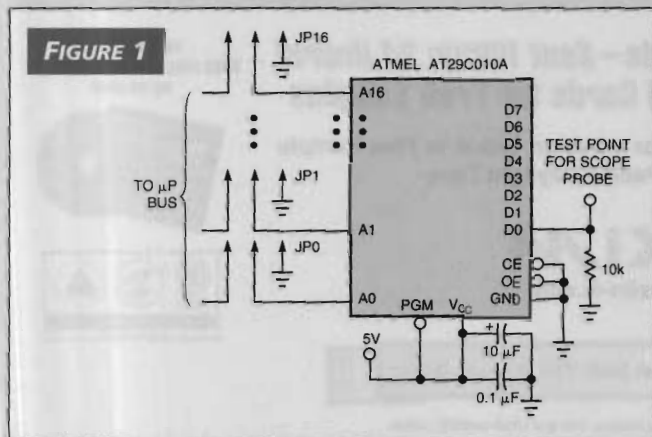
To Vote For This Design, Circle No. 415

DSO-triggering scheme is cheap and efficient

ROBERT PERRIN, Z-WORLD, DAVIS, CA

Although a logic analyzer is useful in troubleshooting a new design, nothing beats a digital storage oscilloscope (DSO) for the ability to see bus levels and timing. However, the trigger mechanisms on most DSOs are not sophisticated enough to

trigger on a specific μ P bus state. For example, you may need the DSO to trigger on a specific address while the μ P is attempting to read from it. This operation would correspond to the start of some code segment you're interested in observing. Most DSOs cannot provide such triggering. **Figure 1** shows a circuit that generates a trigger on a specific μ P bus condition.



A flash chip with all addresses but one filled with zeros dramatically improves DSO-triggering capabilities.

This test pin plugs into a 0.025-in. test point while providing another 0.025-in. pin for other test equipment.

The flash IC serves as a decoder. The entire address space is filled with zeros, except for a single one at the address that corresponds to the μ P bus state of interest. The DSO can trigger on the D0 line of the flash chip, and begin recording events at the bus cycle of interest. Shorting blocks, used with JP0 through JP16, configure the address pins corresponding to the trigger condition. The μ P bus signals required to detect the trigger condition attach to the open address lines in the flash IC. You can program multiple bus conditions to generate a trigger on the same data line. Other data lines can generate additional trigger conditions.

The design in Figure 1 uses a 70-nsec flash chip. The speed is actually irrelevant, because DSOs can acquire and display

pretrigger information. The absolute timing of the trigger is arbitrary. The relative timing of the bus signals is the parameter of interest. The DSO displays pretrigger and posttrigger data properly. As a result, you'll have a solid picture of the bus signals for evaluation. The design in Figure 1 uses a ZIF socket to hold the flash IC; thus, reprogramming is easy. In addition, the test pins in Figure 2 make it easy to grab signal lines from PLCC test clips, while providing a 0.025-in. square post pin to which the logic analyzer can connect. (DI #2197)

EDN

To Vote For This Design, Circle No. 416

Piezo device generates buzz, beep, or chime

DENNIS EICHENBERG, PARMA HEIGHTS, OH

Piezoelectric buzzers, such as the Murata (Smyrna, GA) PKB5-3A in Figure 1, make excellent alarms. They're compact, lightweight, efficient, and reliable. However, a piezo alarm is a dc device; it requires additional circuitry to operate from an ac source. The circuits in Figure 1 provide a simple and inexpensive way to obtain the dc drive. The W04G full-wave bridge rectifier produces a full-wave dc waveform from the 120V ac line. The 100 Ω resistor protects the circuit from surges when you first apply power. The 1N4733 5.5V zener diode protects the buzzer against high-voltage excursions. The 1- μ F capacitor provides filtering for the buzzer.

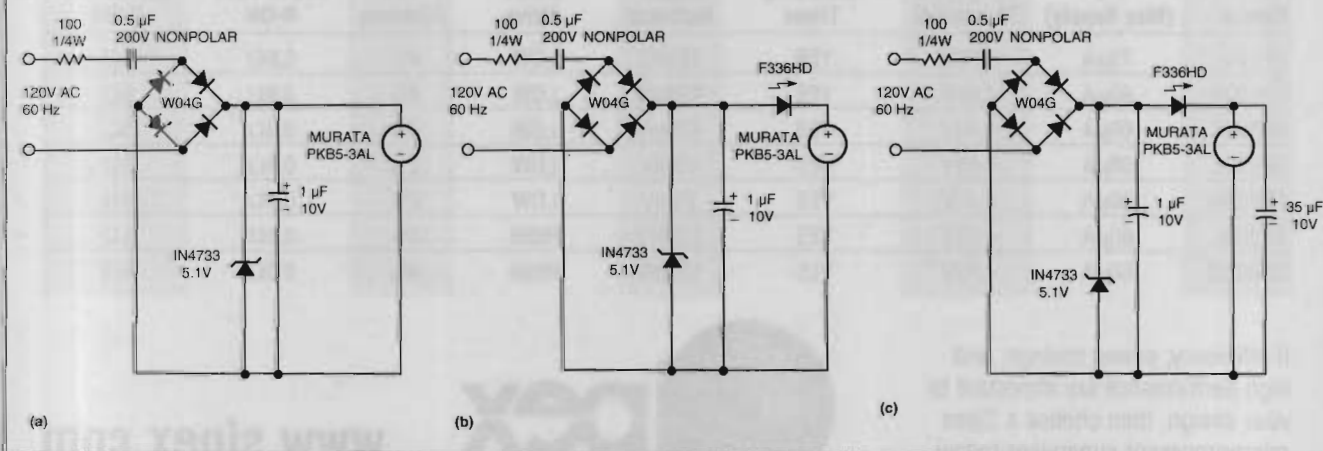
The circuit in Figure 1a produces a true buzzer sound. The

addition of an F336HD flashing LED (part number 276-036 at Radio Shack) in Figure 1b changes the alarm to a beeper, and it also provides a visual alarm. The LED produces a constant pulse of light at approximately 1 Hz without the addition of a time-constant capacitor. The LED starts immediately when you apply power, and it's insensitive to temperature variations. The addition of a 35- μ F capacitor in parallel with the buzzer (Figure 1c) changes the audible alarm to a pleasing chime. The value of the capacitor is not critical; you can obtain various sound effects by varying it. (DI #2194)

EDN

To Vote For This Design, Circle No. 417

FIGURE 1



A handful of inexpensive components configures a piezo alarm device as a buzzer (a), a beeper (b), or a chime (c).